Performance Portability Experiences & Challenges in Lattice QCD

Bálint Joó, Jefferson Lab
discussing also the work of others: P. Boyle (U. Edinburgh), K. Clark (NVIDIA), R. Edwards (JLab), J. Osborn (ALCF), F. Winter (JLab) and S. Khan (ODU) with help from C. Trott (Sandia)

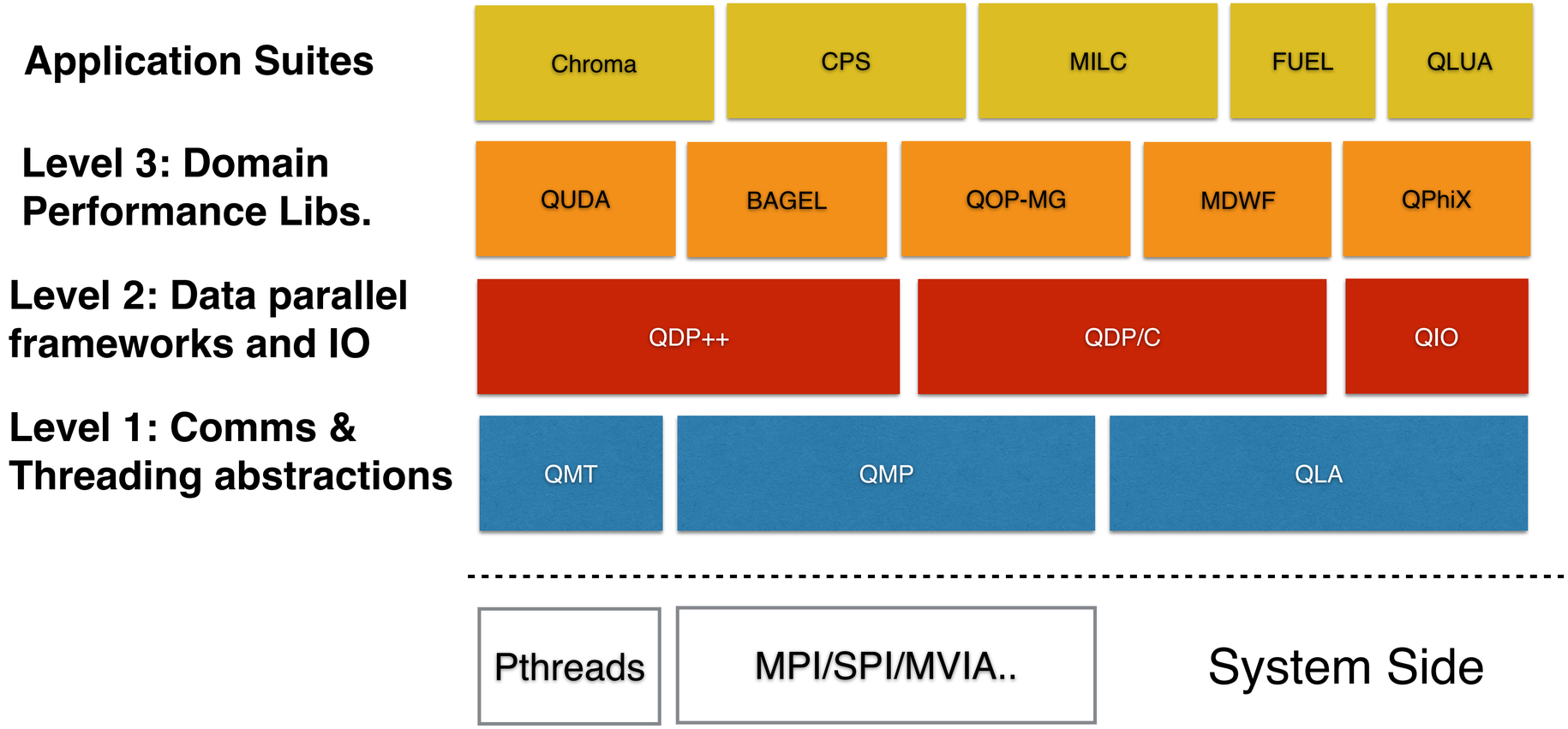
DOE Centers of Excellence Meeting, Glendale AZ
April 19, 2016

Quick Contents

- USQCD Software Layers
  - Pros and Cons
- Bespoke Performance Libraries
- Extreme Metaprogramming
  - QDP-JIT for CPUs and GPUs
  - Grid
  - QEX
- Very recent Kokkos experience
- Lessons Learned

This is a review-like talk, I didn't do all the work. All mistakes/errors etc. about work of others are mine.

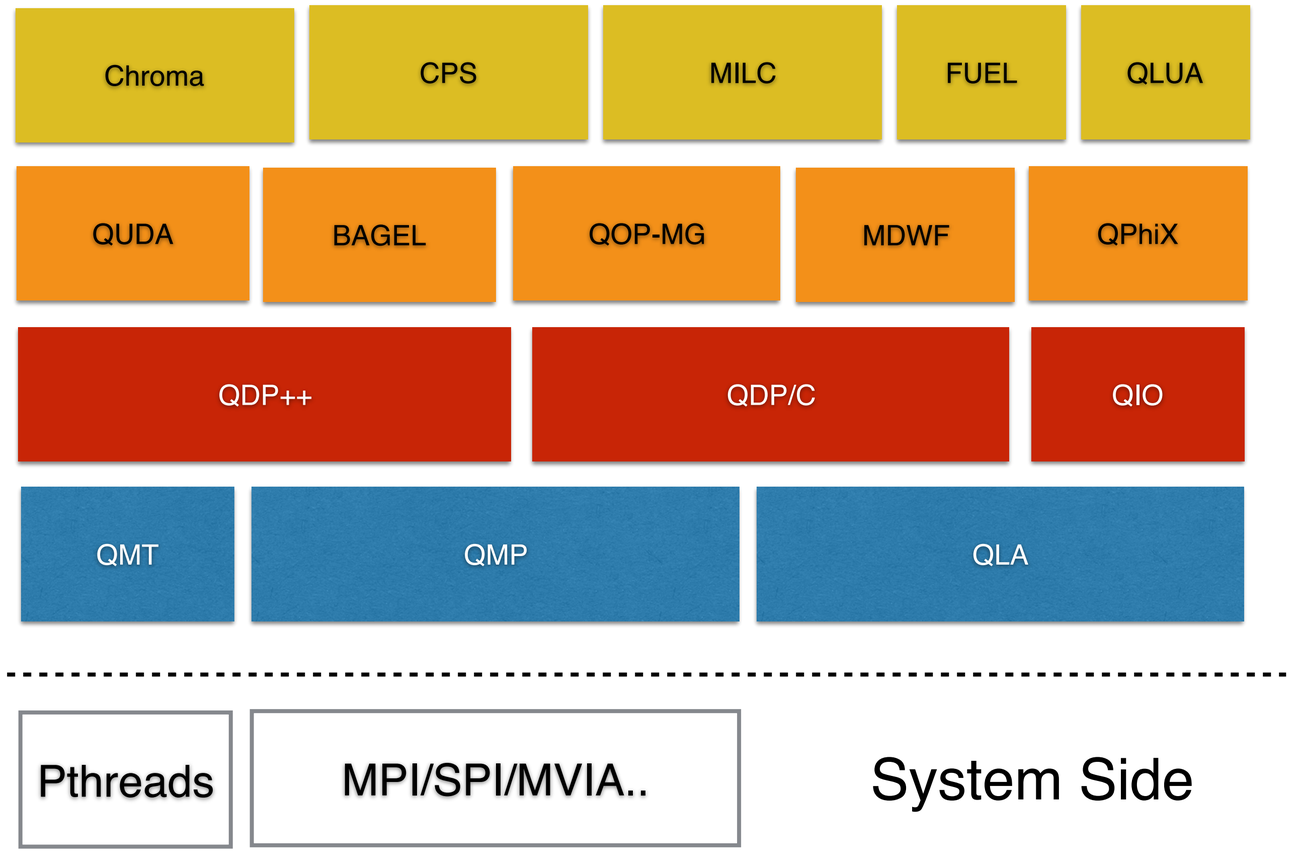
USQCD SciDAC Layered Software



- Layered approach
- Performance libraries - tied to architecture
- Data parallel frameworks for productivity
- Wrap MPI etc.

Pros and Cons

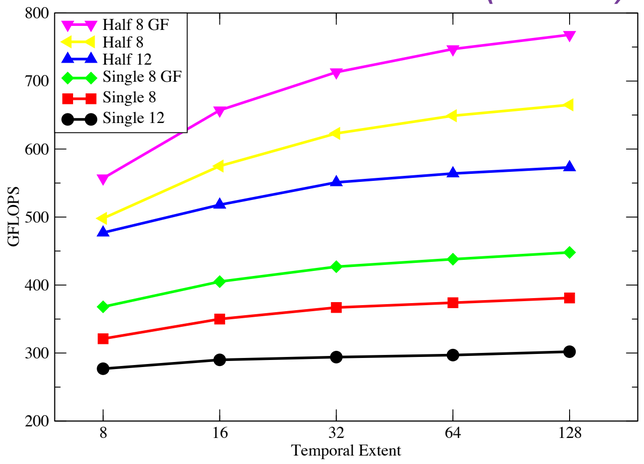
- “Performant Capability Portability”
  - but only as long as capability is provided in Level 3 libraries for all architectures of interest
  - Libraries often compete, features not always in sync
- Good reuse of libraries between applications (e.g. QUDA)
- Has been ‘very portable’
  - QCDOC, Cray XT, XE and BlueGene systems, Xeon Clusters
- Level 2 data layouts have not been flexible, and have typically been AOS. Difficult to vectorize
  - libraries require data import/export, need to be sufficiently granular to amortize this
  - ‘native’ level 2 code, can become Amdahl’s law bottleneck

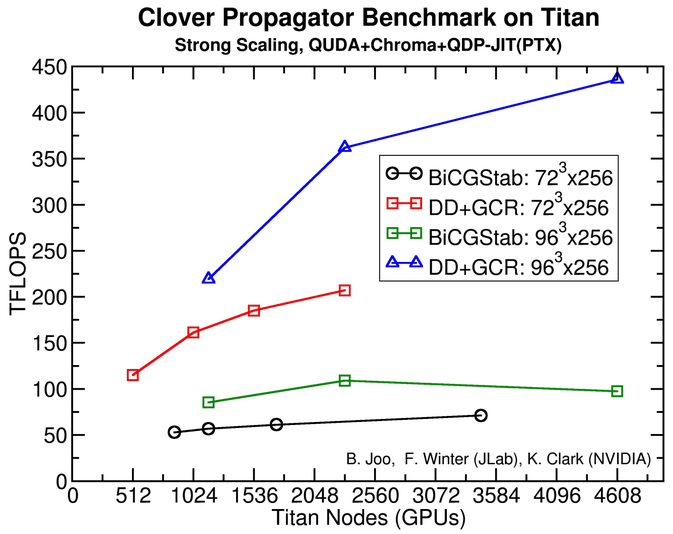


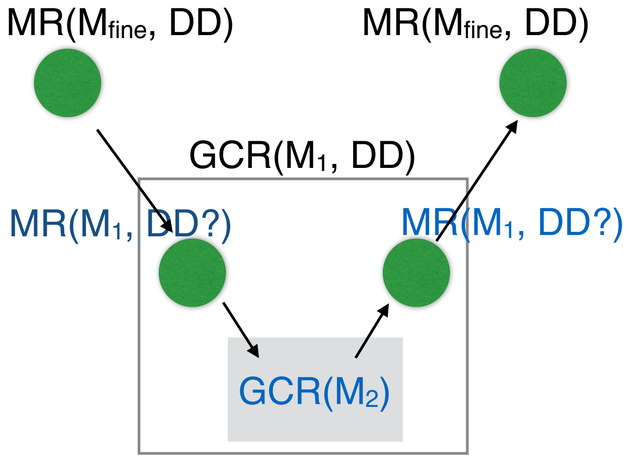
Domain Specific Performance Libraries

- Great way to learn about the architectures
- QUDA for GPUs:
  - vital importance of data layout
  - use of mixed precisions
  - use of compression
  - use scalable methods!
  - CUDA C++ based.
- Bagel and QPhiX libraries for multicore CPUs
  - AOS/SOA/AOSOA layouts
  - cache-blocking approaches
  - load-balancing / prefetching
  - Intrinsics + OpenMP/pthreads based (QPhiX), assembler (BAGEL)
- Very suitable for ‘simple’ solvers (CG & BiCGStab)
- ***Complicated solvers (e.g. multilevel AMG) painful***
  - ***need to refactor library into a framework: e.g. QUDA***

K. Clark - GTC 2016 (K20X)





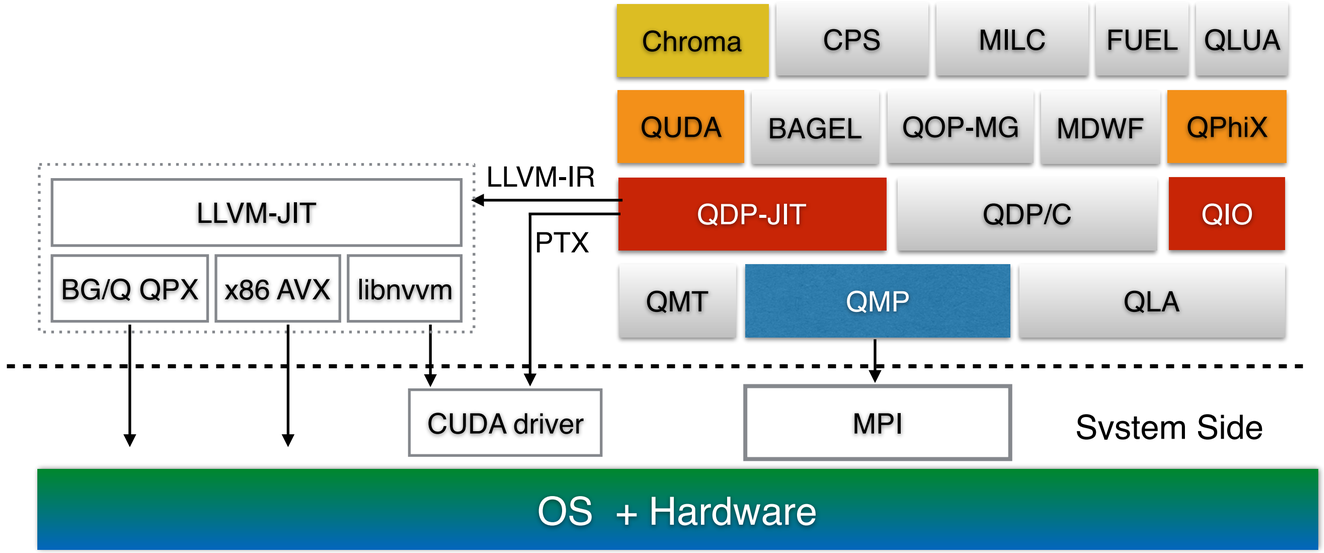


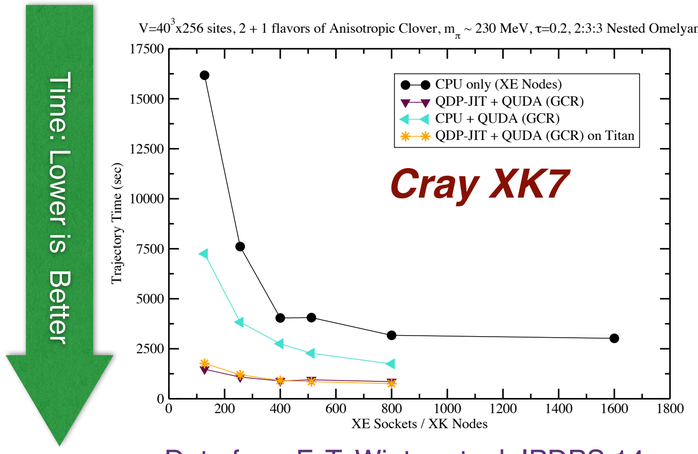
Vcycle Preconditioner:

- multiple solvers,
- indexing multiple lattices
- different operators
  - fine, fine DD?
  - coarse, course DD?
- change layout per level?
- where do grids live?
- where do levels execute?
  - CPU or GPU?

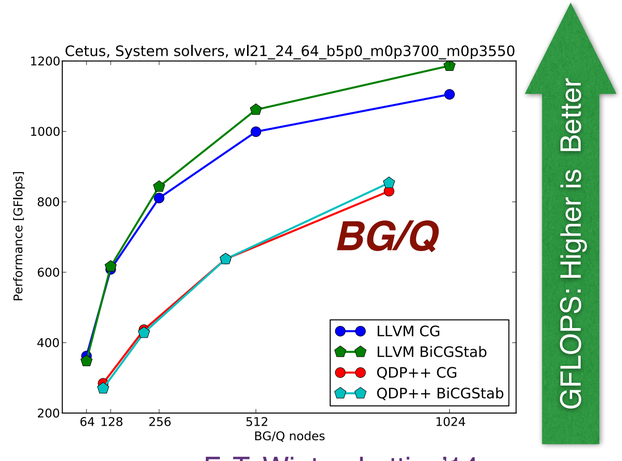
Extreme Metaprogramming I: QDP-JIT

- QDP-JIT
  - Subvert PETE expression templates in QDP++ to produce Just In Time Code Generators
  - Produce PTX (initially) or LLVM-IR (now)
  - Use LLVM optimization passes
  - Use LLVM back end to target hardware
- Drop in replacement for QDP++
  - Deploy all of Chroma on GPUs as well as CPUs
- Divorce data layout from QDP++ data structures
  - different (good) layouts on GPU & CPU
- Treat GPU memory as LRU cache
  - Cache management via CUDA for GPU
  - Could generalize to other Slow/Fast memory combinations.
- Downsides:
  - JIT Overhead: no problem for long duration jobs
  - Dependent on LLVM and PTX versions potentially
  - Maintainance/extension needs LLVM expertise





Data from F. T. Winter et. al. IPDPS 14



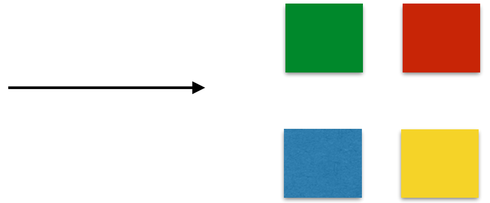
F. T. Winter, Lattice'14

Extreme Metaprogramming II: Grid

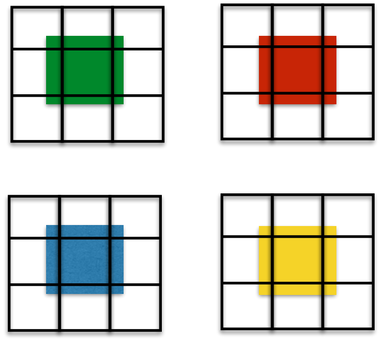
- Recent development from University of Edinburgh (P. Boyle)
  - Address performance concerns of native (non-JIT) QDP++
  - Add multi-level features
  - Emphasis on Data Layout
  - Use virtual node idea from Connection Machine days
  - Solve problem of ‘vectorization’, can work for GPU Warps too.
- Uses C++-11 features for metaprogramming
  - decltype, constexpr etc.
  - smaller code than PETE
  - multi-resoluton ideas (generic forall)
  - comms optimization (e.g. stencil concept)
- Performance is key
  - “Designed to ONLY use and propagate vector intrinsics globally throughout data parallel code: wrapped in high level operators and each defined precisely once in a single short (400LoC) file making porting exceedingly easy”
- Not yet portable to GPUs
  - Future goal: using evolutions of OpenMP offload/OpenACC
  - Combining directives with metaprogramming can be tricky: R&D topic
- See: https://github.com/paboyle/Grid for code, and also Peter Boyle’s slides from IXPUG: [Here] . Look for 06-Boyle-IXPUG.pdf

Vector Unit of Length N

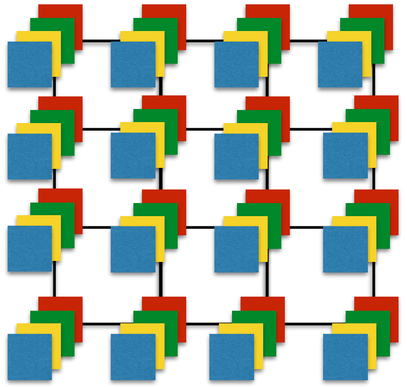
$\log_2 N$ dimensional virtual node grid



Lay-out lattice over virtual node grid



Ascribe corresponding sites from virt. node grid into vector lanes



Extreme Metaprogramming III: QEX

- QEX Project: James Osborn, ALCF, using the Nim language
- Nim (formerly Nimrod) is a very high level language (http://nim-lang.org)
  - generates C/C++, which can then be compiled
    - allows easy integration with C/C++
    - can produce #pragma's in the C-code (OpenMP/OpenACC) intrinsics (SIMD)
    - can interface with C-like languages (CUDA/OpenCL)
  - two levels of metaprogramming
    - use as flexible C/C++ code generator
    - high level AST transformations on any Nim code
  - strongly typed
  - modules: no more arbitraty .h/.cc distinction
  - reflection/introspection: automatic (de)serialization
  - script-like: instead of scripting your executable, have your script be the executable
    - no need to integrate scripting/input language
- Comments:
  - relatively young language
  - would require some infrastructure work (e.g. wrap MPI — Nim provides automatic wrapping tool)
  - very promising initial exploration work
- See http://www.mcs.anl.gov/~osborn/scidac/qex.pdf for details
- QEX code available on GitHub: https://github.com/jcosborn/qex

From J. Osborn's slides: Example Lattice (client) Code

```
import qex
import qcdTypes

qexInit()
var lat = [4,4,4,4]
var lo = newLayout(lat)
var v1 = lo.ColorVector()
var v2 = lo.ColorVector()
var m1 = lo.ColorMatrix()
threads:
  m1 := 1
  v1 := 2
  v2 := m1 * v1
  shift(v1, dir=3, len=1, v2)  # len=+1: from forward
  single:
    if myRank==0:
      echo v2[0][0]  # vector "site" 0, color 0
qexFinalize()
```

Construct Domain Specific Objects

Matrix x Vector
Nearest Neighbors

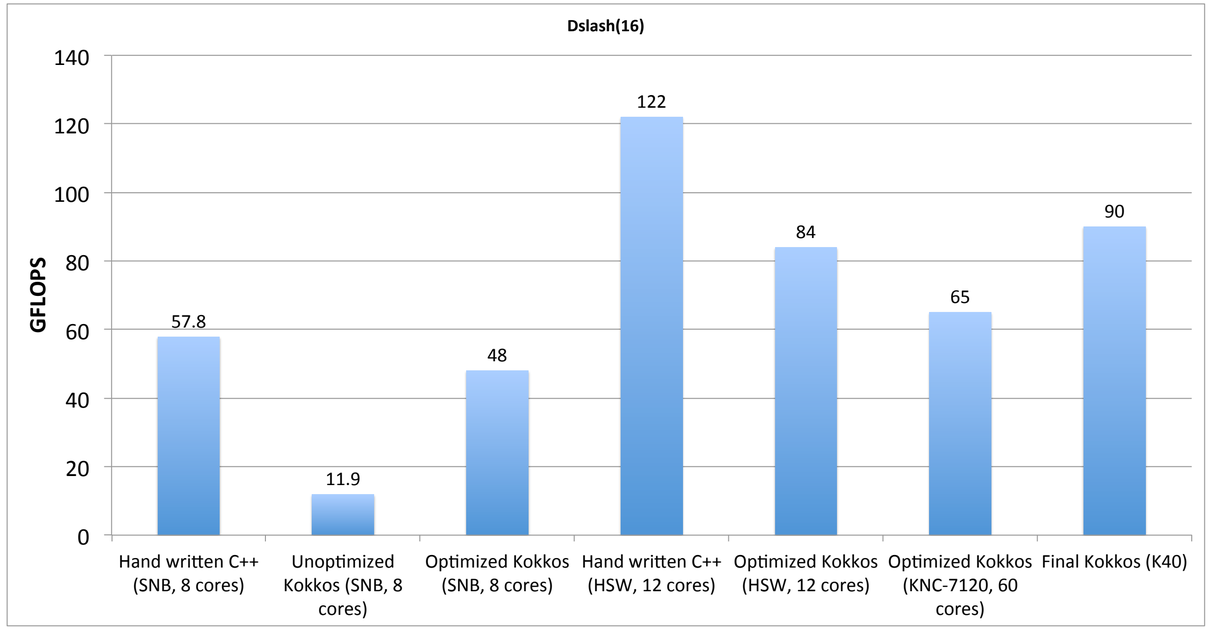
```
template threads*(body:untyped):untyped =
  let tidOld = tid
  let nidOld = nid
  proc tproc =
    {.emit:"#pragma omp parallel".}
    block:
      setupForeignThreadGc()
      tid = ompGetThreadNum()
      nid = ompGetNumThreads()
      body
  tproc()
  tid = tidOld
  nid = nidOld
```

Emit Compiler #pragma

From J. Osborn's slides: Metaprogramming example

Kokkos Experiments

- Kokkos is a programming model for performance portability written in C++
  - many back ends: OpenMP, CUDA, qthreads etc.
- Aesthetically very appealing abstraction of hardware features, and implementation of parallel patterns.
- Initial experiment with a lattice QCD Kernel: $y_i = D\, x_i$ for i=0..15
  - compare Kokkos implementation with a straight C/C++ version. (S. Khan, ODU)
  - Initial performance low (no vectorization)
  - Interaction with Kokkos team (C. Trott) improved performance to over 80% of hand written C++ code
- Big Thanks to Christian Trott from the Kokkos team for optimizing this so promptly!!!
- More work to be done here, hopefully future collaboration with Kokkos team.
- Kokkos is on GitHub: https://github.com/kokkos
  - also, further talks at this workshop: C. Trott, S. Hammond, & others



NB: Optimized = what Christian optimized overnight
Final = what Christian optimized overnight + 1 day
This is not the end, but the beginning (custom layouts etc still to come)

Lessons Learned/Takeaways

- Bespoke High Performance Libraries are extremely valuable but also a real pain
  - High performance, but tied to architecture
  - Low level code, using intrinsics/assembly.
  - Frequently generated by code-generators
  - Complicated algorithms (more complex solvers, force terms) are difficult to implement
    - turn ‘library’ into a ‘framework’ from the bottom-up
  - Feature compatibility often lags between libraries
  - Difficult to maintain
  - Low developer productivity
- Would really like the productive Level 2 frameworks to provide features and performance to remove the need for bespoke libraries
  - performance portability through the performance portability of the framework
  - but potentially decreased reuse for codes using different frameworks (framework dependence)

Lessons Learned/Takeaway: II

- Data layout flexibility and mixed precision algorithms are crucial, especially for memory bandwidth bound problems
- In order to provide ‘expression’ syntax for users, and in order to deal with GPUs as well as CPUs - we had to do to some form of metaprogramming
  - PETE - for QDP++ & QDP-JIT; C++11 improvements for Grid
  - Kokkos uses a lot of metaprogramming under the hood
  - Mixing compile time constructs, with runtime constructs can get very messy.
    - e.g. runtime selection of templated objects
- Nims offers nice features (modules, reflection, scripting etc) and can be used as a high level code-generator for C/C++.
- We’ve not made much use of directives for accelerators so far
  - QUDA library used CUDA since beginning
  - QDP-JIT works at the PTX/LLVM-IR level (depending on implementation)
  - There may be issues using acceleration directives with recursive expression templates.
  - Frameworks should hide this feature from the user.
- Essential to have clean builds, unit tests etc
  - Code is complex (Expression Templates, Low level codes etc), Testing is important. We should always strive to improve software engineering practices.

Acknowledgements

- Many thanks to colleagues for rapid & constructive feedback
  - P. Boyle (U. Edinburgh), K. Clark (NVIDIA), R. G. Edwards (JLab), J. Osborn (ANL, ALCF), F. Winter (JLab)
- Many thanks to C. Trott of the Kokkos team for extremely fast optimization of our test code
- I gratefully acknowledge funding from the US Department of Energy, Office of Science, offices of Advanced Scientific Computing Research, Nuclear Physics and High Energy Physics under the SciDAC program
- This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics under contract DE-AC05-06OR23177.